Mapping Packet Processing Applications on a Systolic Array Network Processor

Olivier Morandi*, Fulvio Risso*, Pierluigi Rolando*, Olof Hagsand† and Peter Ekdahl‡
* Politecnico di Torino, Italy
Email: {olivier.morandi, fulvio.risso, pierluigi.rolando}@polito.it
† Royal Institute of Technology (KTH), Sweden
Email: olofh@kth.se
‡ Xelerated AB, Stockholm, Sweden
Email: peter.ekdahl@xelerated.com

***Abstract*—Systolic array network processors represent an effective alternative to ASICs for the design of multi-gigabit packet switching and forwarding devices because of their flexibility, high aggregate throughput and deterministic worst-case performances. However such advantages come at the expense of some limitations, given both by the specific characteristics of the pipelined architecture and by the lack of support for portable high-level languages in the software development tools, forcing software engineers to deal with low level aspects of the underlying hardware platform.**

In this paper we present a set of techniques that have been implemented in the Network Virtual Machine (NetVM) compiler infrastructure for mapping general layer 2-3 packet processing applications on the Xelerated X11 systolic-array network processor. In particular we demonstrate that our compiler is able to effectively exploit the available hardware resources and to generate code that is comparable to hand-written one, hence ensuring excellent throughput performances.

I. Introduction

Handling packets at wire-speeds in the order of tens of gigabits per second and more, while still ensuring some degree of flexibility, cannot be afforded without relying on massively parallel architectures because the processing cost in network processing is usually dominated by the high latency of memory operations. In fact, memory technology has not kept the pace with the speeds required on backbone networks. For example, on an OC-768 line at 40 Gbps, a 40 bytes packet should be processed in at most 8 ns, i.e. the time needed for performing at most a couple of memory accesses to an off-chip SRAM. In order to reduce the impact of the latency of memory on the processing time, some kind of task parallelization should be put in place, e.g. by pooling several processing elements and connecting them by one or several high-speed interconnection devices, or by arranging them in a linear pipeline [1].

Systolic-array network processors extend such considerations to the extreme by giving new life to the concept of pipelined dataflow architectures, which have already been proved successful in the field of Digital Signal Processing. Essentially, systolic architectures include a deep pipeline composed of hundreds of processors, each one usually able to perform a single VLIW operation. A packet entering the pipeline is processed by the first stage, then it moves to the second stage while another packet is ready to be processed by the first one; therefore packets progress synchronously at fixed intervals, until they reach the end of the pipeline. In particular, for a pipeline composed of n stages, the maximum allowed throughput is proportional to $\frac{1}{t}$, where t is the time needed for performing a single VLIW operation, while the overall latency (i.e. the time needed for a packet to traverse the entire pipeline) is proportional to nt, which is usually a negligible value at line rates in the order of tens of gigabits per second.

The major advantages of using systolic array network processors come as a consequence of the previous considerations, i.e. the deep pipeline allows high throughput, while the synchronous advance of packets in the pipeline guarantees deterministic worst-case performances. In fact, commercial NPUs based on such kind of technology, like the Procket PRO/Silicon [2], the Bay Microsystems Chesapeake [3] and the Xelerated X11 [4] ensure wire-speed processing at line-rates ranging from 10 to 40 Gbps, or even 100 Gbps, while still operating at clock rates in the order of hundreds of megahertz.

However, the promised advantages come at some costs in terms of ease of development of software applications. In particular, the difficulties are mainly tied to the inherent characteristics of the pipelined architecture and the lack of tools providing a high level abstraction of the hardware. Indeed, software engineers usually have to program the packet processors using some kind of assembly language, and even when higher level tools are available, such as a modified flavour of the C language, they are forced to deal with the details of the target architecture.

In this paper we present the architecture of a compiler based on the NetVM [5] model, which is able to generate code for the X11 NPU, as well as a set of techniques for exploiting the available hardware resources, with results that are comparable to hand-written code. This result is especially interesting because it denies the common belief that achieving high performances with network processors requires writing programs by hand using assembly language. In addition, this result helps demonstrating the benefits of the NetVM model, which provides a flexible, efficient, and hardware-independent framework for writing packet processing applications. More specifically, applications written using this model can be executed on different hardware, can support data-link to

application-layer processing, yet are capable of high speed.

The paper is structured as follows: Section II reports the available related work, while Section III gives an overview on the architecture of the Xelerated X11 network processor and on the characteristics of the NetVM programming model, in Section IV the architecture of the compiler is described, while Section V outlines the implemented mapping techniques. Experimental results are reported in Section VI, and conclusions are drawn in Section VII.

II. Related Work

The compilation of computation-intensive programs on systolic array processors was first explored in [6], however the problem of mapping packet processing applications on systolic array network processors is novel and still relatively unexplored.

The programmability of network processor architectures is a topic that has been widely discussed by the research in recent years. In [7] a C compiler for an industrial network processor was proposed, showing that exposing low level details in the language through intrinsics and compiler known functions allows an efficient exploitation of the available hardware features without relying on assembly language, while [8][9][10] describe novel domain specific languages, programming models and compilers for automatically partitioning packet processing applications on multi-core based network processors.

The proposed solutions are very target-specific because they tend to expose the features available on the target hardware to the programmer; while they are proven to work well with the chosen architecture (e.g., a multi-core network processor), it is not clear how effective they would be when applied to the generation of code for systolic array network processors. The Network Virtual Machine (NetVM) [5] improves previous solutions because it provides an abstraction layer based on a sequential programming model in which hardware is virtualized, with the result of completely hiding the target architecture to the programmer, while still allowing an efficient mapping.

III. Background

A. The X11 Network Processor

The Xelerator X11 network processor is based on a systolic array (actually a pipeline) with a synchronous dataflow architecture, which shares the concept of a systolic pipeline with its predecessor X10q [11].

Figure 1 shows an overview on its internal architecture.

The processing elements are either VLIW processors called Packet Instruction Set Computers (PISCs) or I/O processors called Engine Access Points (EAPs). As shown in Figure 1a PISCs are arranged in blocks while EAPs are placed at fixed points between PISC blocks. EAPs essentially dispatch the computation to special purpose devices that can be used to offload part of the computation off the PISC pipeline. Such devices include TCAMs, counters, hardware for computing hash values, external SRAM, etc.

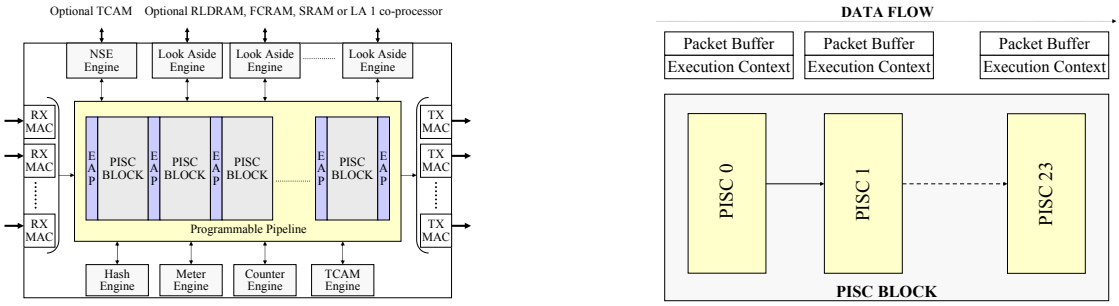
When a packet enters the pipeline, it is first partitioned into fixed size fragments. Thereafter, the pipeline processes the packet fragments using iterations of (1) PISC processing interrupted by (2) actions and look-ups orchestrated by EAPs. As a fragment traverses the pipeline, it carries an individual execution context containing the fragment itself, a register file, status registers, and other information that constitute the complete state of a program. Figure 1b shows the details of a PISC block. It is important to understand that one PISC acts on one packet fragment during exactly one cycle. During this cycle, the PISC can perform a set of parallel instructions on the fragment, before passing it on to the next element in the pipeline.

The parallelism of the pipeline is hardwired in the architecture itself. From one perspective, this makes the software handling of concurrency easy, since the execution contexts and PISCs are effectively isolated from each other. No explicit mechanisms such as threads or mutexes need to be adopted to protect accesses to these local resources. It is also easy to access external resources as long as this is made in a constrained fashion, primarily limited by the look-up bandwidth towards external engines.

However, generic update of shared state is difficult to realize due to pipeline hazards, including Read-After-Write, Write-After-Write, etc. [12]. The reason is that the non-stalling nature of the synchronous pipeline makes it impossible for a program to wait indefinitely for an asynchronous mutex. However, for the X11, a mutex mechanism can be achieved by looping or by controlling the traffic scheduling into the systolic pipeline. If no hardware-provided mechanisms exists, all such shared accesses need to be scheduled in advance when configuring the pipeline for a specific application. Fortunately, the X11 architecture offers some means for providing more elaborate accesses to shared resources. This includes support for atomic read-and-increment operations both on the on-circuit counters engine as well as external RAM locations.

From the compiler perspective, a X11 packet program consists of a number of instruction sequences that are laid out in the instruction memory of the pipeline. This memory is actually a two-dimensional matrix with rows and columns where the control flows unidirectionally and synchronously between columns, and branching occurs between rows. Because of the unidirectional execution flow, loops are not possible by definition; branches, however, are allowed. The layout of code in the instruction memory can be seen as a two-dimensional optimization problem, where a vertical column constitutes the instruction space of a single PISC, and the horizontal rows are instruction sequences. The execution context contains a row instruction pointer so that PISCs know which instruction to execute. Branching modifies the row instruction pointer but does not affect the horizontal flow of the program.

The drawbacks to this programming model are tied to its advantages. First, looping is not allowed: programs requiring loops need to be unfolded to some limit that fits the pipeline. The X11 also provides a loopback path to let packets reenter the pipeline if the program is longer than the number



(a) X11 Architecture

(b) PISC Block Detail

Fig. 1. X11 Internal Architecture Overview

of pipeline stages allows. The number of pipeline passes, k, is statically configured at link-time and is limited since the throughput is proportional to $\frac{1}{k}$. The operating frequency of the X11 systolic pipeline is dimensioned to allow a specific number of loops while still providing wire-speed. Moreover, in order to avoid reordering, all packets coming from the same input interface always undergo the same number of pipeline passes, even if the processing could terminate earlier for some of them.

Second, there are few methods to share state between packets. In particular, it is difficult for information from one packet to influence the processing of another. This includes programs that adapt to traffic contents traffic, e.g., stateful packet filters. To provide for shared state between packets, one can use the support from the existing counter engine or implement some other, more elaborate mechanisms in the general-purpose look-aside engines. It is also possible to communicate with the control plane, which in turn can re-program the pipeline by altering the state of look-up tables, but this approach has the obvious drawbacks of being limited in bandwidth and also may introduce race-conditions.

B. The Network Virtual Machine

The Network Virtual Machine is a programming model for writing packet processing applications, e.g., the ones running on the data plane of a network device. This model provides an abstraction layer that hides the differences among hardware devices (e.g., network processors), thus enabling the paradigm "write once, run anywhere" for packet processing applications as well. Notably, flexibility (the model supports applications that range from data-link to application layer) and portability are not obtained at the expense of performance.

The NetVM model does not define any high-level programming language because these are usually tailored to specific applications; instead, it defines a mid-level abstraction layer (called the Network Intermediate Language, NetIL) that can be employed as a target for several high-level programming languages, either declarative (e.g. rule-based packet filtering and classification languages) or imperative (e.g. C-like languages). This allows the NetVM to be general enough to support several classes of packet processing applications (possibly written in different languages) while still allowing the generation of efficient code.

In the NetVM model packet-processing programs are expressed through the interconnection of a set of modules called Network Processing Elements (NetPEs), virtual processors that execute NetIL instructions. A NetPE is composed of a set of private registers and a hierarchy of memories, such as a local memory for storing state information local to a processing engine and an exchange memory for storing the packet buffer along with additional metadata.

Since NetPEs are programming abstractions, the execution of a NetVM program on real platforms relies on the existence of a full optimizing compiler (either JIT or AOT) that maps the abstract resources of the NetVM model onto the ones available on the real platform, i.e., a translator of NetIL instructions into native machine code of the target hardware architecture.

Packet processing applications are usually built upon a core set of primitives that are often implemented directly in hardware on many network processor architectures (e.g. Content Addressable Memories for fast table look-ups) so the NetVM model includes the concept of virtual coprocessors, a way for making such features available to the programmer through a well-defined interface. A coprocessor is seen by the application as a black box providing specific operations; although its coherent interface guarantees the portability of the software using it among different platforms, its implementation varies from platform to platform. In particular, coprocessors must be emulated in software by the run-time environment of the NetVM on architectures that do not provide any hardware acceleration, while they are mapped directly on architectures that provide the required special purpose features.

The intrusion detection sensor presented in [13] (a Snort clone) demonstrates that the NetVM model supports complex applications; particularly, that example uses 11 interconnected NetPE modules and 3 virtual coprocessors. More details on the NetVM abstraction can be found in [5].

IV. NetVM Compiler Infrastructure

The architecture of the compiler for the NetVM platform, depicted in Figure 2a, follows the classical design of a multi-target optimizing compiler, apart from operating on a bytecode representation of the application instead of using a high level

source language. Besides, in order to support different target architectures, the compiler is structured so that most of the phases involved in the compilation process are common to all the targets, while the phases involved in the generation of code for a specific platform are isolated in different back-ends. The overall compilation process is structured as follows: (1) the program is first translated into a more manageable tree-based intermediate representation (IR) and its formal correctness is verified, (2) a set of target-independent optimizations are applied on the IR, then finally (3) the optimized program is transferred to the selected back-end, which handles all the subsequent target-specific transformation tasks.

The mid-level optimizations are mostly classical ones, such as constant folding, constant propagation, dead code elimination, control-flow simplification and algebraic reassociation [14]; they have the twofold purpose of reducing the impact of redundant and useless code on the program size, as well as enabling further optimizations and special purpose transformations, as will be detailed in Section V.

A. A Back-end for the X11 NPU

In order to provide a mapping for NetVM applications on the X11 processor, a new back-end for the NetVM compiler has been developed. Its architecture is depicted in Figure 2b.

The back-end translates the tree-based intermediate representation generated by the upper layers of the compiler into a low-level IR (i.e. a representation of the program very close to the assembler of the target machine), while mapping the accesses to virtual coprocessors on instructions that make use of the special purpose hardware features (e.g. TCAMs) available on the target architecture. This task is performed by a Bottom Up Rewriting System [15], which executes a tree-matching algorithm driven by architecture-specific rules that specify how a portion of the intermediate representation (i.e. an expression subtree) should be translated into target instructions. If multiple rules relate to overlapping tree patterns the BURS is able to chose the best (i.e. less expensive) combination that covers the most extended expression tree. As will be detailed in Section V-D, this enables the deployment of advanced optimizations because recognizing specific patterns of instructions that represent macro functionalities allows their mapping directly to hardware coprocessors or special purpose instructions.

In contrast to traditional processors, the X11 NPU completely lacks the concept of function call; therefore a NetVM application composed of multiple NetPEs must be transformed into a single compilation unit to be laid out as a linear code sequence throughout the PISC pipeline. The X11 back-end compiler addresses this problem by performing an inlining step in the compilation process, where the code belonging to different NetPEs is linked together by replacing inter-module calls with jump instructions. This inlining operation is possible only if the NetPE interconnection graph is acyclic, however this property is intrinsically ensured by the NetVM model.

Afterwards, the intermediate representation is further optimized by removing redundant instructions that might have been generated during the instruction selection phase, then the resulting code is examined to detect independent instructions that are suitable to be merged in VLIW blocks. At the end of the compilation process, a resulting assembly file is created which can be used as an input for the X11 SDK tools that create the proper binary files for loading and execution.

V. The Mapping Process

Compiling a packet processing program for the X11 NPU does not differ significantly from compiling it for any other kind of processor, as long as only the generation of sequences of target instructions from high level constructs is considered. However, some constraints that are specific to systolic architectures, along with some characteristics of the X11 processor, suggest the adoption of specific compilation techniques in order to best exploit the available hardware resources and to improve the chance of a program to be correctly and efficiently compiled.

This section explores the major problems related to the efficient mapping of NetVM applications on the X11 architecture and presents the most innovative aspects of the NetVM compiler infrastructure.

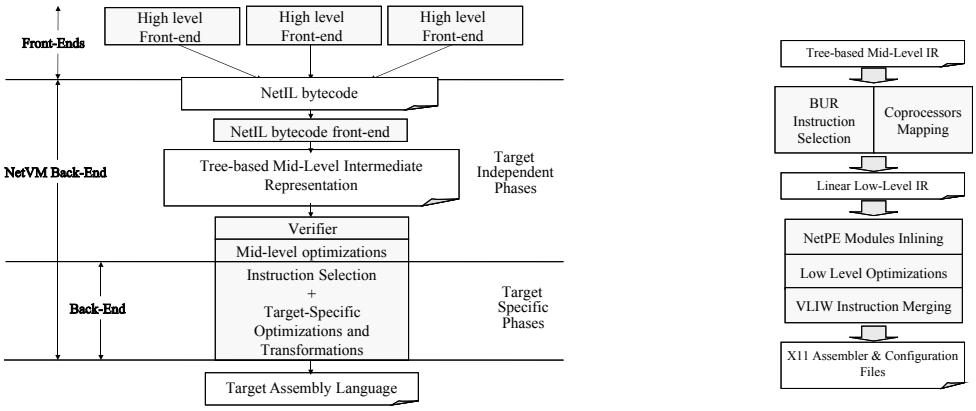
A. Handling Loops

Since backward pointing branch instructions are forbidden, systolic array processors are characterized by an ”upstream to downstream” execution model, where the control flow is driven by data flowing through the pipeline and cannot be redirected to a previous stage. This translates to the impossibility of mapping generic loops on a systolic array, unless their maximum number of iterations is bounded and known at compile time, so that they can be completely unrolled and laid out as a linear sequence of instructions. However, even in this case some practical problems arise: the theoretical upper bound on the number of iterations may be so large that the resulting overall instruction count could exceed the number of available stages even when using the loopback path as described in Section III-A.

If such considerations apparently pose a strong limitation on the kinds of applications that can be successfully and efficiently mapped onto a systolic array network processor, it should be noted that uncontrolled loops are not frequent in standard forwarding programs (either L2 or L3) with the exception of some protocols (e.g., MPLS stacking or IPv6 extension headers [16]). In such cases the problem can be overcome by limiting the maximum number of loop iterations in the source program to a fixed value. Such considerations point out that the theoretical limitation of systolic arrays in handling loops may not be so relevant in practice.

B. Keeping the State of the Application

The NetVM model uses different memories to keep the state of an application. In particular, state information local to a NetPE is stored in the NetPE local register file and local data memory, the former keeping temporary values while the latter is used for static values as well as complex structures.



(a) NetVM Compiler Infrastructure

(b) X11 Back-end

Fig. 2. Compiler Architecture

Vice versa, the state that is local to a packet is stored in the packet buffer and a special buffer called the ”info memory”, i.e. a memory segment that allows subsequent NetPEs to communicate between them.

On the X11 side, the execution context is represented by the packet memory and a register file, while persistent state must be kept in externally attached memories that are accessed through the EAPs. As a matter of fact there happens to be a significant parallelism between the NetVM model and the X11 processor when it comes to data associated with a packet. In particular, the X11 packet memory and register file allow indirect addressing and can be used to map the NetVM packet buffer and the info memory. Besides, a portion of the register file can be allocated for keeping intermediate results as they are computed in the NetPEs, as well as local register values.

On the other hand, the two platforms differ in the way permanent data (i.e. the state that survives across different packets) is treated. As detailed in Section III-A, there are constraints on how multiple, concurrent accesses to the same external memory location can be made. Section V-D reports how in very specific cases the compiler is able to handle this problem while still ensuring the safe update of shared memory locations.

C. Mapping NetVM Coprocessors

The NetVM model allows complex operations and functionalities to be represented as invocations to virtual coprocessors. The back-end maps them on the corresponding hardware features (if available on the physical device) in order to maximize the efficiency of the resulting code. In particular, on the X11 processor this usually translates to generating instructions that send and receive data to/from the EAPs and declaring which engine operation should be performed.

A look-up coprocessor that allows the programmer to associate 32-bit keys to 32-bit values was considered as a proof of concept. On the X11 the requested operation can be performed by the integrated TCAM module. Since the same hardware unit must possibly be shared with other instances of the same coprocessor (in a different NetPE) or contain other unrelated content, the compiler provides a thin hardware abstraction layer to split the TCAM into multiple tables. This is achieved by dedicating a portion of the look-up key space to hold a table number.

NetVM coprocessors wrap a well-defined interface around a usually complex algorithm; the compiler has the twofold task of translating the algorithm itself and adapting the interface to the actual hardware units employed. While the latter task is achieved by the compiler, the former might prove impossible due to possible limitations of the hardware platform. In particular, if the target architecture does not provide the specific functionalities exposed by a virtual coprocessor, a software emulation must be performed. However, this might not always be possible due to restricted amount of primitives provided by the hardware and limitations imposed on the instruction count.

D. Exploiting the Features of the Hardware Architecture

The previous section explored the problem of mapping a virtual coprocessor (i.e., a specialized macro-functionality) defined by the NetVM model on real hardware. This section presents the dual problem, i.e., mapping generic NetIL code to some specialized modules provided by the hardware.

Apart from the case where the source language exposes high level constructs that find a natural mapping on specific hardware functionalities, the problem is in general extremely complex: hardware modules usually implement complex algorithms that, in order to be efficiently translated, must first be recognized in the source program.

The *switch-case* provides a simple example of an easily recognizable high-level construct. The instruction count of a traditional implementation based on a linear search might grow in complexity with the number of possible destinations, potentially using an extensive portion of the pipeline. However, on the X11 the same behaviour can be obtained by performing an associative look-up that uses the on-board TCAM, costing effectively one pipeline stage only, independently from the number of possible alternatives.

An unintended consequence of extensively using this mapping technique might be the over-subscription of limited

hardware resources. In particular, there are limitations in look-up bandwidth and also the fact the EAPs are present at specific stages in the pipeline. In this case the compiler should emit code that uses other pipeline resources such as the PISC processors or different external units. Although deciding when to do this is a complex optimization problem, the compiler tries to solve it through a simple heuristic that works well in the average case.

Making the specialized functionalities provided by the X11 hardware automatically available to the program requires in the general case more effort than mapping the *switch-case* construct. A good example derives from the problems related to the concurrent update of shared information mentioned in Section V-B. If the state to be updated is an integral value, the compiler can make good use of the X11 support for atomic increment instructions, thus becoming able to overcome concurrency issues in a limited set of cases. A common example is keeping counters in external memory, e.g. for statistical purposes.

A counter increment operation in itself is not atomic as it is necessary to fetch the old value, increase it and store the newly computed result at the same offset. However if this procedure is not performed atomically by the hardware it becomes possible for two consecutive packets to read the same value from memory with the net effect of incrementing the counter once instead of twice. To overcome this issue the compiler uses the BURS-based instruction selector which is able to recognize if specific locations of the data-memory are accessed through this pattern of operations, and to map them on the special purpose atomic increment instructions provided by the hardware.

Depending on how the source code is written, it can happen that a pattern ends up split across different statements. Since the BURS operates on a single IR expression tree at a time, in this case the recognition mechanism does not work. No control on the source code form can be assumed, so this issue would result in low reliability of the compilation process if left unchecked. Vast improvements can be made by processing the intermediate representation with appropriate optimization algorithms, such as algebraic reassociation. These algorithms can rearrange subtrees in the IR so that the semantic meaning of the program is preserved, but providing the instruction selector with deeper trees that are more likely to contain recognizable patterns. This way the BURS can operate successfully even if the related instructions were originally scattered across a region of the source listing.

In any case, it must be pointed out that even though such techniques work well in very specific cases, their general validity still needs to be proven, since they are tuned on patterns of instructions and not on algorithms. In particular, even for the simple example of counters, the programmer could update a specific memory location in several exotic ways, preventing the BURS to recognize the sequence of instructions as a predefined pattern. We believe that in order to deploy a general algorithm recognition technique, more specialized analyses of the code should be performed.

E. VLIW Instruction Merging

Being VLIW processors, PISCs allow up to four independent operations to be executed at the same time, in order to exploit instruction-level parallelism. These can be (1) an ALU operation, (2) a move for copying words of up to 32 bits between different locations of the register file and the packet memory, (3) a load offset operation for indirectly accessing the register file or packet data, and (4) a branch.

When generating assembly code, the compiler should try to merge multiple instructions in single VLIW words, taking care appropriately of data and control dependencies. Several algorithms are described in literature for handling such task in an optimal way, e.g., trace scheduling [17]. The compiler currently implements a simple algorithm that works only on straight-line code fragments (i.e., basic blocks) and does not perform any instruction reordering before merging. This provides good results, even though it is a widely known result that the amount of instruction-level parallelism present in a program is limited when considering only basic blocks, even more if instructions are never reordered. It is likely that implementing a more aggressive strategy would improve the emitted code quality significantly.

F. Automatic Computation of Data Size

While the NetVM model allows to fetch and store any data size ($\leq$ 32 bits), registers are 32-bit words. This is a problem for the X11 processor that works natively on 16-bit words because of the larger overhead required to perform 32-bit operations, while often these can be correctly carried out using only 8 or 16 bits.

Although this is clearly a limitation of the NetVM model that does not explicitly support different data sizes, we decided to implement an heuristic algorithm in the X11 back-end that tries to assign to each NetVM register the optimal, minimum size while preserving the program semantics. In the long term, this issue points out the necessity of a revision of the NetVM model that will involve the addition of new NetIL opcodes to provide the NetVM with hints about the appropriate data size.

VI. Experimental Results

The X11 architecture presents many properties that make it predictable, allowing to exactly determine the behaviour of a program through off-line statical analysis, without runtime benchmarking. The reason is that throughput is constant and proportional to $\frac{1}{t}$, as long as the code fits into the instruction memory of the systolic pipeline. Therefore, if the code is proven correct, a useful evaluation metric is the amount of instructions generated by the compiler. With a fixed size pipeline and a given number of passes, translating a program to fewer instructions allows more features to fit in the program with the same deterministic throughput. Additionally, if the instruction count is comparable to the one of hand-written code, the compiler proves to be useful and claims that systolic arrays NPUs must be hand-coded can be rejected.

In the evaluation, two test programs were used: *(i)* a Snort [13] module that performs L2-3-4 packet inspection and

saves data for subsequent modules, and *(ii)* a simple packet filter that demultiplexes and counts TCP packets directed to port 80. Although these applications are small, we claim that the operations they perform are rather common in packet processing programs and stress several NetVM capabilities, using coprocessors and several kinds of memory.

Since there are currently no other optimizing compilers for the X11, it is hard to the get the baseline results needed to evaluate the performance of the NetVM compiler. To get relevant results the source programs were first translated with all optimizations turned off. A second compilation was performed on the same source files, with all the automatic optimizations enabled. Afterwards the code, as already optimized by the compiler, was further processed by hand to apply a wider range of transformations, using standard optimization guidelines used by Xelerated. The same procedure was repeated keeping the VLIW merging algorithm disabled in order to better appreciate its impact on the resulting code size.

Results are shown in Figure 3: the ones related to the IDS module are on the left (Figure 3a), while the ones related to the filter application are on the right (Figure 3b). Both the total number of instructions are shown, as well as the number of resulting VLIWs after instruction merging. As it can be seen, the number of instructions for the Snort application is $87/76$ for the automated and hand-written cases respectively, while the corresponding numbers for the filter application is $23/19$. After instruction merging, the results were $68/48$ for the Snort module and $22/17$ for the filtering.

Current results are encouraging: even with a prototype compiler and small applications, the instruction count obtained with the compiler is within 20% of the size of hand-optimized code before VLIW merging. Moreover, this was obtained by a proof-of-concept code that often used simple algorithms to speed up the implementation. We believe production-quality code can push this result even more. The differences between manual and automatic optimizations can be mainly ascribed to the simple VLIW merging algorithm employed, that does not perform instruction reordering, and to some missed copy folding opportunities. Both these issues can be addressed with standard techniques described in literature that do not require a redesign of the compiler framework to be implemented. Finally, it is worth noticing that the VLIW merging is not a fundamental module for the objectives of this paper.

A second evaluation objective is to identify what kinds of packet processing applications can be compiled into X11 code. In this area it is more difficult to present tangible results. However, we have focused on the primitives that the NetVM and its compiler make available on the X11 platform, trying to ascribe any limitations to either the NetVM model itself or to the target architecture, that in turn limit the class of applications that can be compiled.

The result is that simple L2-L3 applications can be implemented since all the required primitives are supported by the compiler. The range of supported applications can be extended to L4 if the application is stateless.

On the contrary, it is more difficult to translate applications that need to keep complex state information such as those using data memory in the NetVM model. Simple cases (e.g. counters) can be handled directly because there are appropriate provisions in the X11 processor but others, including TCP session tracking tables, are more difficult to map, although it is still possible to implement these more complex applications using other mechanisms. Choices range from performing updates in a control-plane context where it is possible to ensure that no concurrent conflicting operations happen to introducing more elaborate special-purpose external engines. However, such solutions are beyond the scope of this paper.

VII. Conclusions

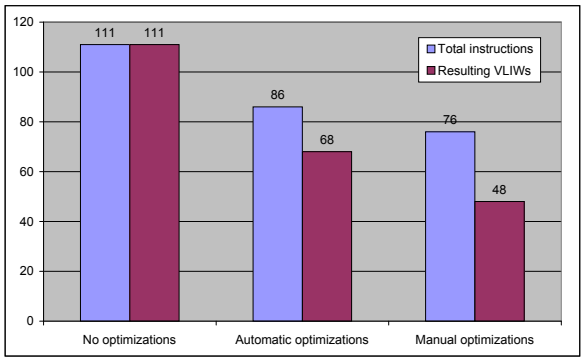
This paper presents a compiler that is able to generate packet processing code for the Xelerated X11 network processor.

The primary objective of this work was to address the problem of mapping L2/L3 packet processing applications on a systolic array processor, without using native assembly and without penalties in flexibility and performances. The secondary aim was to investigate the generality of the NetVM model and its capability to represent an adequate abstraction layer for porting applications on different network processors. Particularly, the synchronous systolic pipeline architecture of the X11 NPU is radically different from the ones of more traditional processors, therefore the ability of successfully mapping on it the NetVM abstraction gives a good insight about the validity of the model.

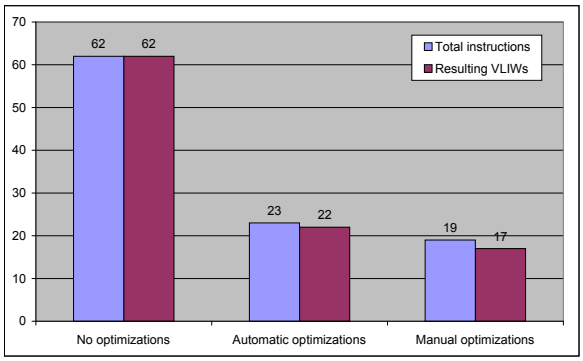
In this work, we extended the NetVM compiler infrastructure with an X11-specific back-end, so that existing packet processing programs expressed in NetIL can be compiled into low-level code that can run on the X11 processor. In our results, we have proven that several, although simple, programs can be generated with only about 20% overhead compared to hand-coded methods.

Among the problems encountered in the mapping, a general issue concerns loops. Due to its structure, the systolic pipeline disallows the presence of generic loops in a program, although our approach provides a method for dynamically detecting a sub-class of loops whose number of iterations is known at compile time and unfolding them within the PISC stages. Unbounded loops, or loops with a high iteration count are not supported. Moreover, we claim that for the packet forwarding applications at hand, this limitations may actually not be so serious as it might appear; for instance [13] demonstrates that it is possible to write a complete L7 application without loops. Furthermore, it is often possible to impose a practical limit to the loop unrolling also in case of theoretically unbounded loops.

A second issue concerns accessing shared state. The non-stallable nature of systolic pipelines makes the implementation of mutual exclusion mechanisms different from traditional architectures. Certain types of atomic operations on shared resources are made available by the X11 hardware, and are successfully exploited by the compiler in order to enhance the probability of source programs being mapped to the target processor.



(a) IDS module



(b) TCP filter

Fig. 3. Code size for the test programs.

From a programmability viewpoint, it is still an open issue what packet processing applications can be executed by a systolic array such as the X11. The work presented in this paper focused on relatively straightforward L2/L3 applications. However, in order to be able to manage functions such as TCP session tracking, the back-end would have to be extended e.g. with the capability to generate code for the X11 control plane, or to make use of ad-hoc external engines.

The NetVM model showed a problem in handling variables with different data sizes. In particular the X11 is a 16-bit processor, while the NetVM abstraction relies on 32-bit variables. The compiler uses an heuristic in order to infer the size of variables and registers, but a more robust approach would involve an extension of the NetVM model itself. Another issue was related to the by-design limit of modelling data-plane functionalities only, which left some operations performed by the control plane out of the mapping.

An interesting result is related to the BURS system, which is shown to be able to recognize patterns of instructions and map them appropriately. However it can hardly cope with the more general problem of recognizing whole algorithms in the source code. In this respect some more investigations are needed and may be objective of future work.

On the positive side, the compiler was able to map the NetVM model nicely, including virtual coprocessors, and it was able to efficiently exploit the hardware resources (e.g., on-circuit TCAM and native counters) not originally envisioned in the NetVM model, guaranteeing performance and portability, thus disproving the common belief that NPUs must be coded in assembly by hand.

Concluding, given the limitations that we have detected we consider our work a success in showing that (1) it is possible to construct a compiler that generates efficient code for a systolic array network processor; (2) with some modifications, the NetVM model is general enough to be mapped on a radically different packet processing architecture.

REFERENCES

[1] J. Fu and O. Hagsand, "Designing and evaluating network processor applications," in *Workshop on High Performance Switching and Routing (HPSR 2005)*, 2005, pp. 142–146.

[2] C. Systems. Cisco systems to purchase intellectual property, engineering teams and select assets from procket networks, press release. [Online]. Available: http://newsroom.cisco.com/dlls/2004/corp_061704.html

[3] B. Microsystems. Chesapeake network processor. [Online]. Available: http://www.baymicrosystems.com

[4] Xelerated. Xelerator X11 network processor. [Online]. Available: http://www.xelerated.com

[5] M. Baldi and F. Risso, "Towards effective portability of packet handling applications across heterogeneous hardware platforms," in *Proceedings of the 7th Annual International Working Conference on Active and Programmable Networks*, November 2005.

[6] T. Gross and M. S. Lam, "Compilation for a high-performance systolic array," in *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*. New York, NY, USA: ACM, 1986, pp. 27–38.

[7] R. Wagner, J.; Leupers, "C compiler design for a network processor," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, no. 11, pp. 1302–1308, Nov 2001.

[8] R. Ennals, R. Sharp, and A. Mycroft, "Linear types for packet processing," in *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, Barcelona, Spain, March 2004, pp. 204–218.

[9] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-la: achieving high performance from compiled network applications while enabling ease of programming," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 224–236.

[10] G. Memik and W. Mangione-Smith, "Nepal: A framework for efficiently structuring applications for network processors," 2003. [Online]. Available: citeseer.ist.psu.edu/article/memik03nepal.html

[11] J. Carlstrom and T. Boden, "Synchronous dataflow architecture for network processors," *IEEE Micro*, vol. 24, no. 5, pp. 10–18, 2004.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[13] O. Morandi, P. Monclus, G. Moscardi, and F. Risso, "An intrusion detection sensor for the netvm virtual processor," September 2007, technical Report TR-DAUIN-NG-02, September 2007.

[14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[15] C. W. Fraser, R. R. Henry, and T. A. Proebsting, "Burg: fast optimal instruction selection and tree parsing," *SIGPLAN Not.*, vol. 27, no. 4, pp. 68–76, 1992.

[16] O. Morandi, F. Risso, M. Baldi, and A. Baldini, "Enabling flexible packet filtering through dynamic code generation," September 2007, technical Report TR-DAUIN-NG-01, September 2007.

[17] J. A. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers*, vol. 30, no. 7, pp. 478–490, July 1981.